

# Cours de compilation

Christophe Alias

# Objectifs

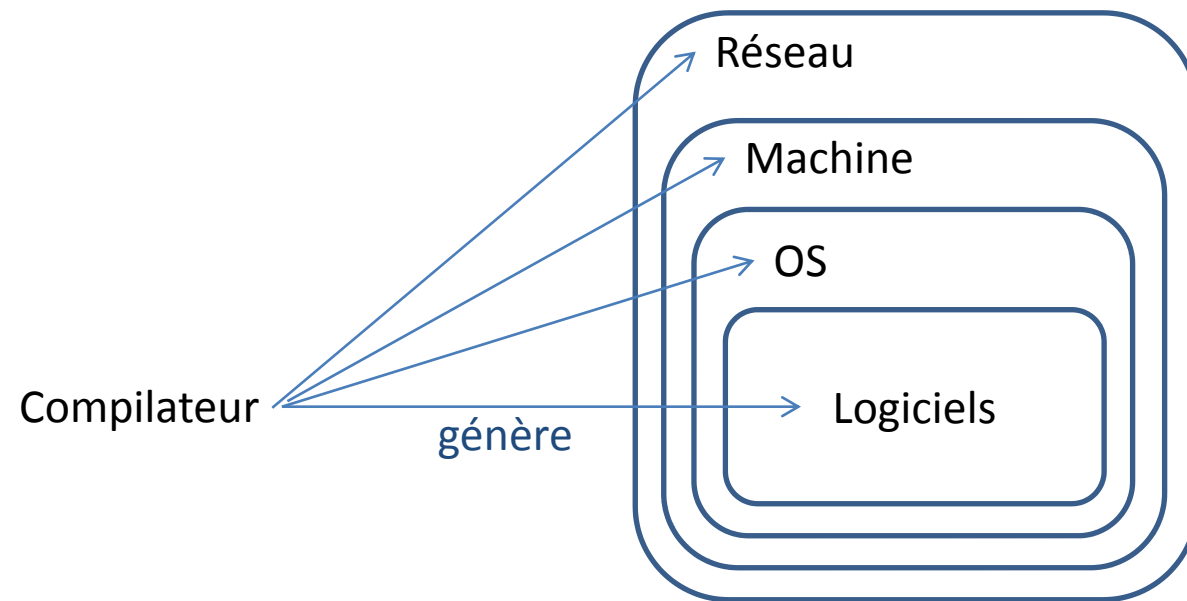
- Etudier le fonctionnement d'un compilateur  $C \rightarrow x86\_64$
- Modèles, algorithmes et outils pour traduire un programme en code machine correct et efficace.
- **Compétences:**
  - Développement/maintenance d'un compilateur
  - Optimisation/débogage avancé

# Contenu du cours

1. Compilateur “simple”
2. Compilation des fonctions
3. Compilation des données structurées
4. Optimisation de code
5. Génération de code machine

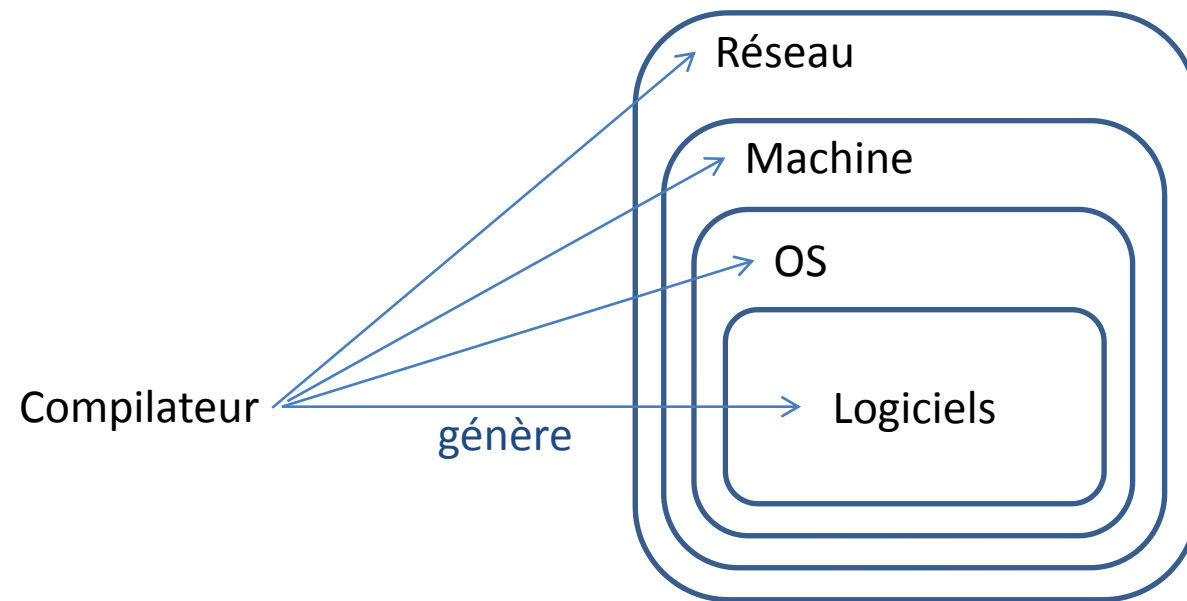
# Pourquoi étudier les compilateurs?

- Parce qu'il n'y aurait pas d'informatique sans compilateur!



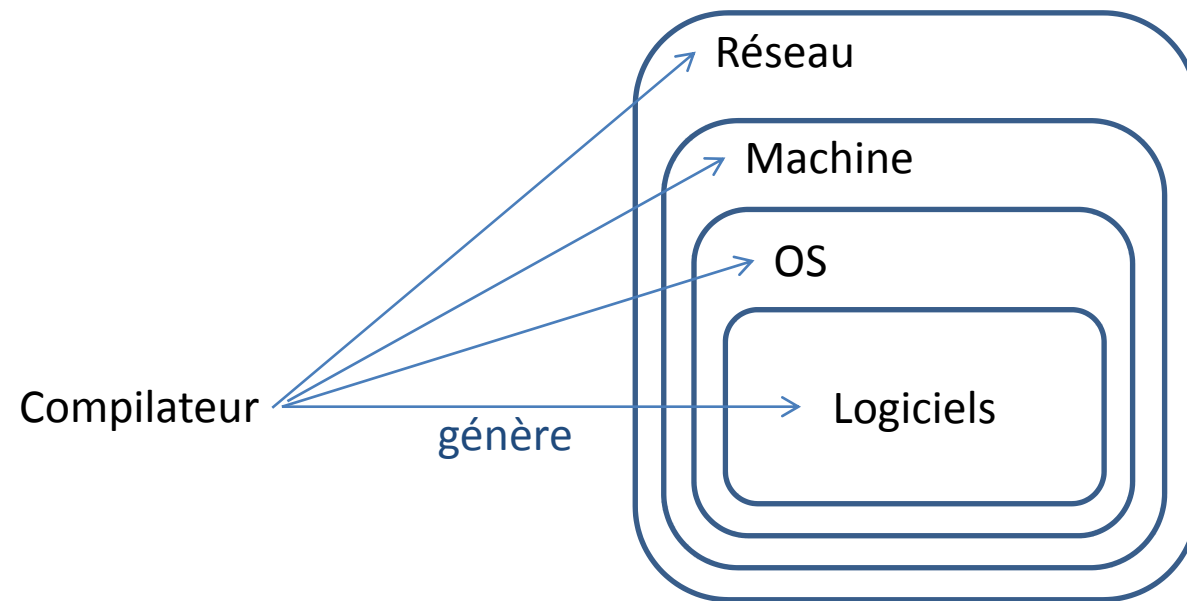
# Pourquoi étudier les compilateurs?

- Le compilateur détient le pouvoir!
- Pas de vraie **sécurité informatique** sans garanties sur le compilateur

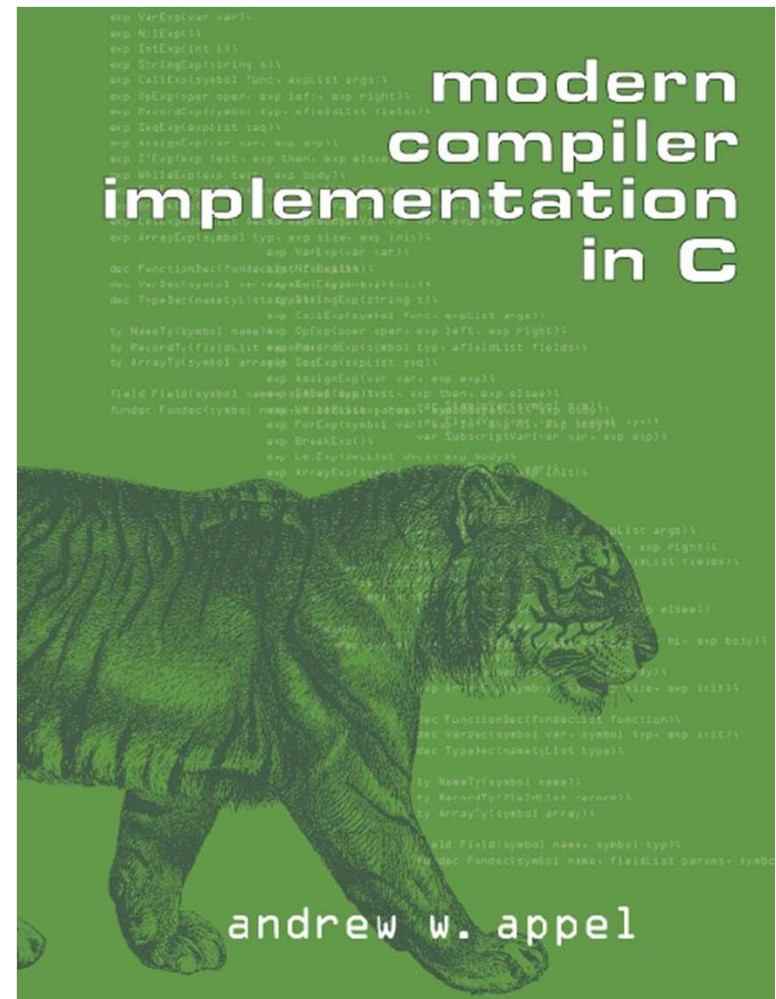
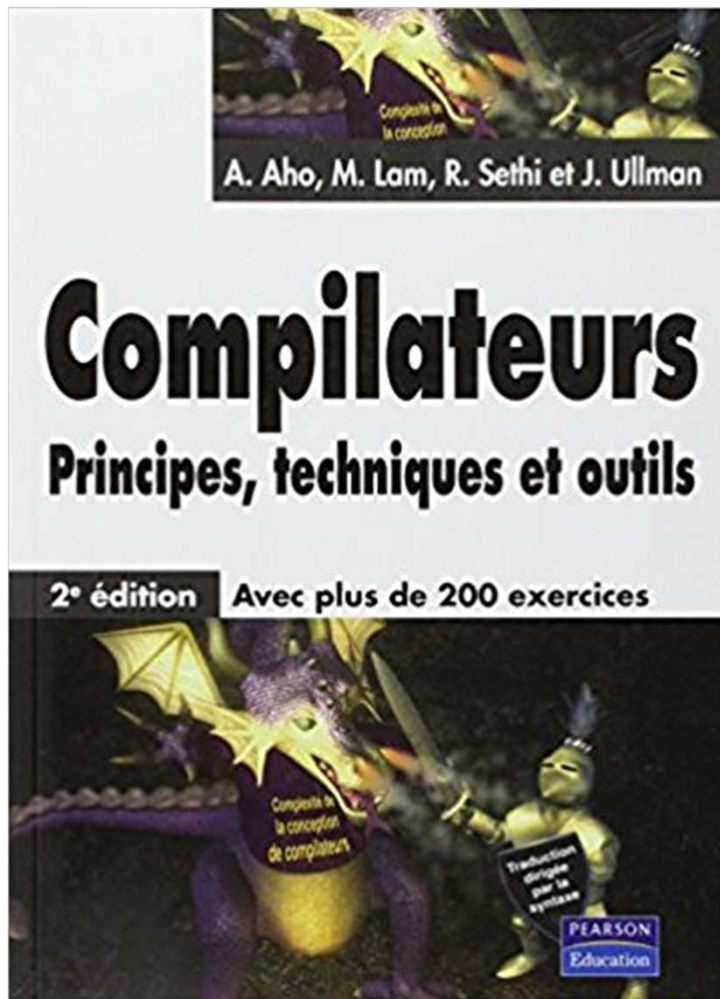


# Pourquoi étudier les compilateurs?

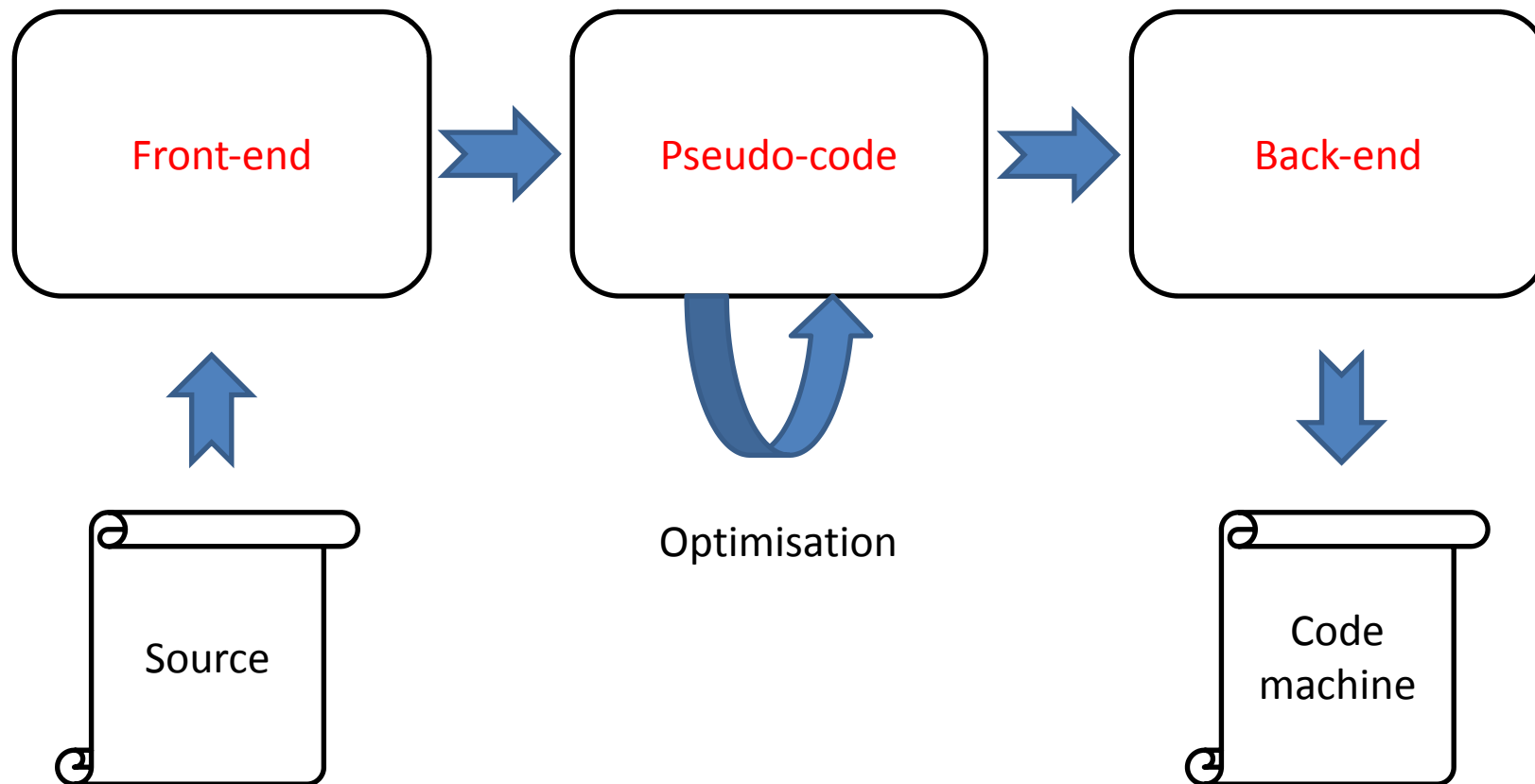
- incorrect: système **bogué** et **indéboguable**
- inefficace: système **inefficace** et **inoptimisable**



# Livres

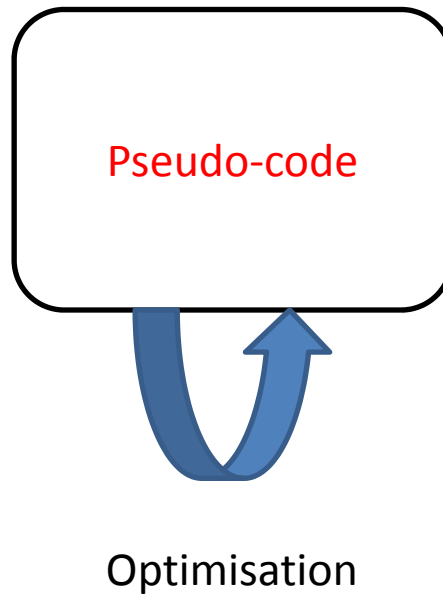


# Etapes de la compilation



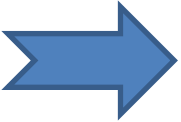


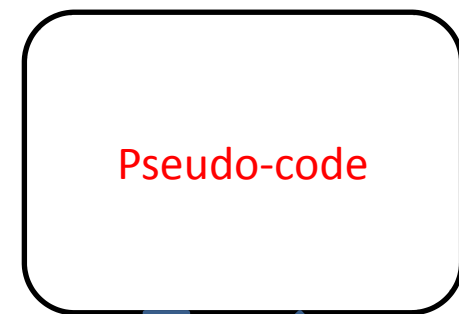
# Quel modèle de machine?



# Quel modèle de machine?

- Machine à pile
  - Variables et résultats intermédiaires sur une pile
  - Génération de code très simple
  - Back-end non-standard

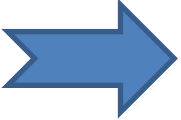
$y = -1 * x;$   `mpush -1  
push x  
mul  
pop y`

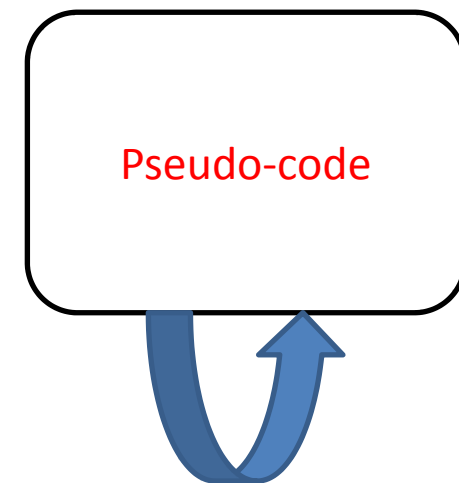


Optimisation

# Quel modèle de machine?

- Machine à registres
  - Variables et résultats intermédiaires dans des registres (en nombre infini)
  - Code plus difficile à générer
  - Back-end standard

$y = -1 * x;$    $t0 = -1$   
 $t1 = [x]$   
 $t2 = t0 * t1$   
 $[y] = t2$

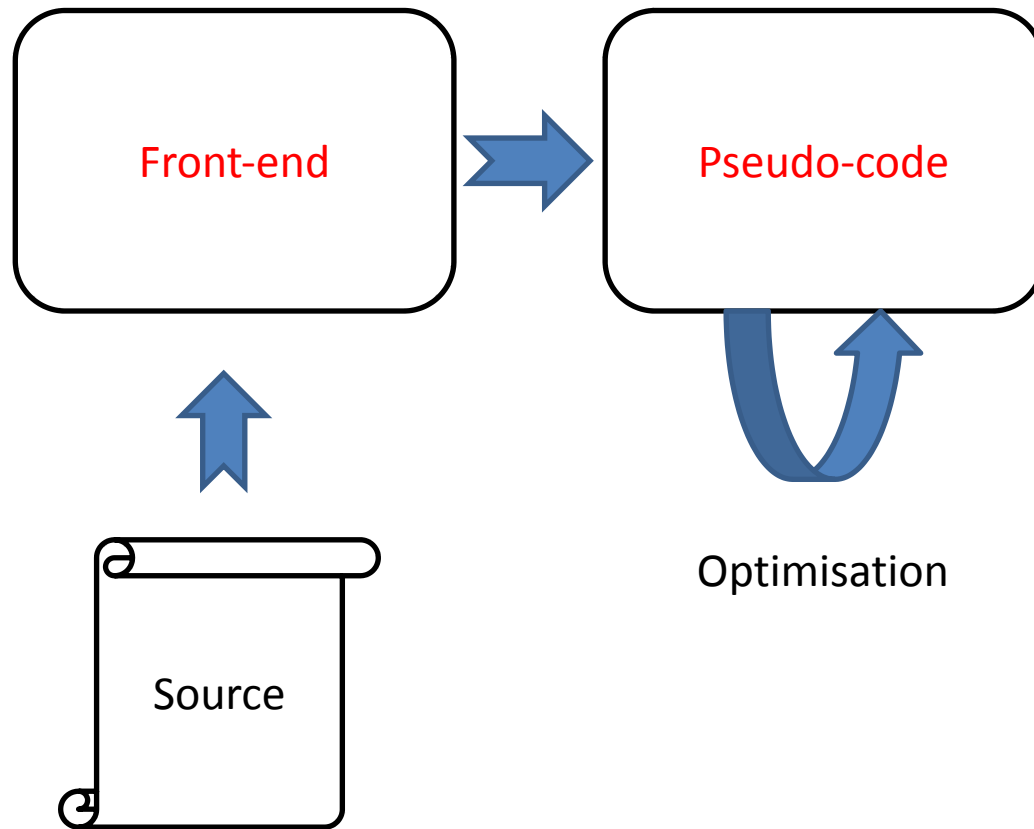


Optimisation

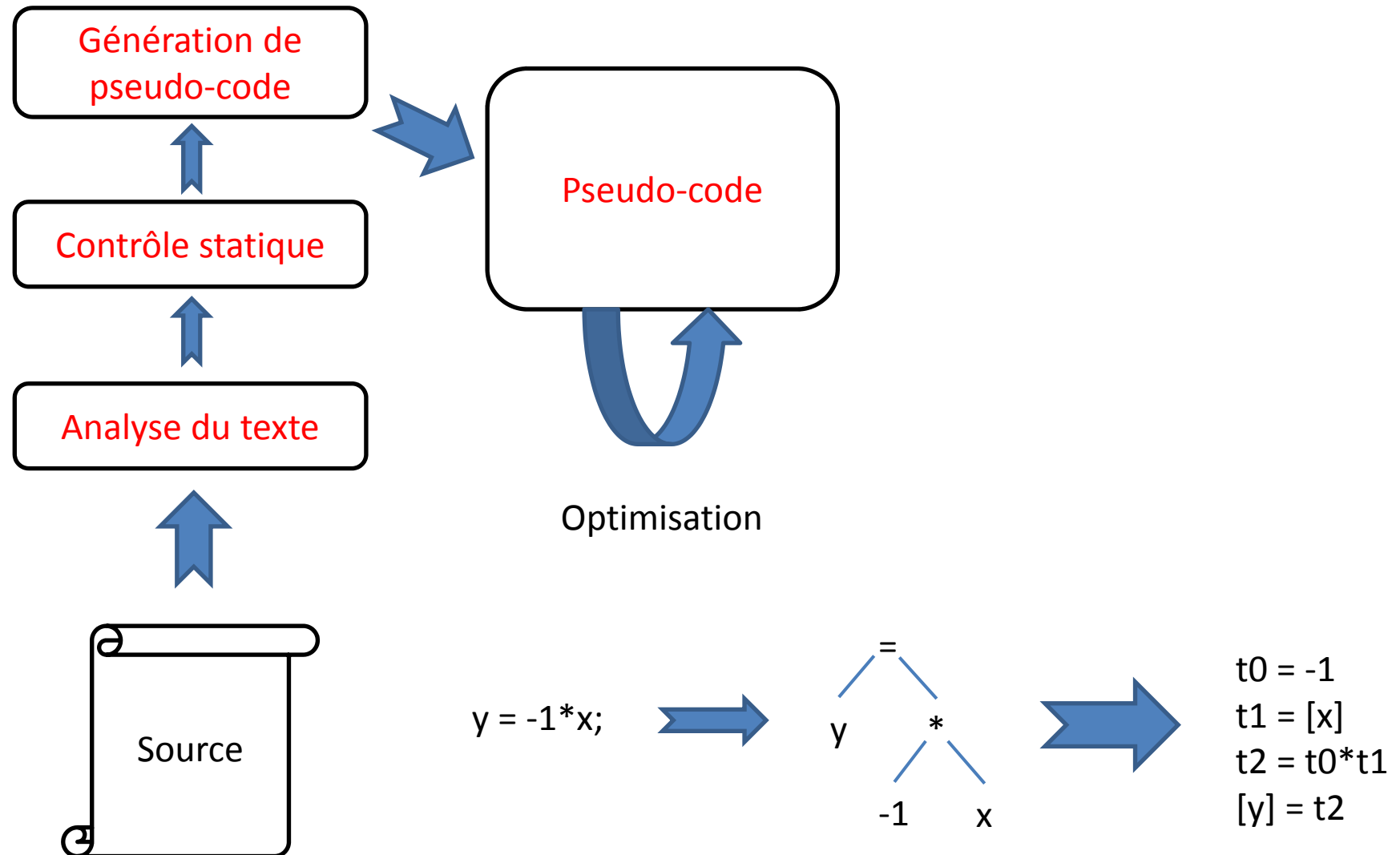
# Quel modèle de machine?

- Compilateur “simple”
  - Compilation des fonctions
  - Compilation des données structurées
  - Optimisation de code
  - Génération de code machine
- 
- machine à pile
- machine à registres

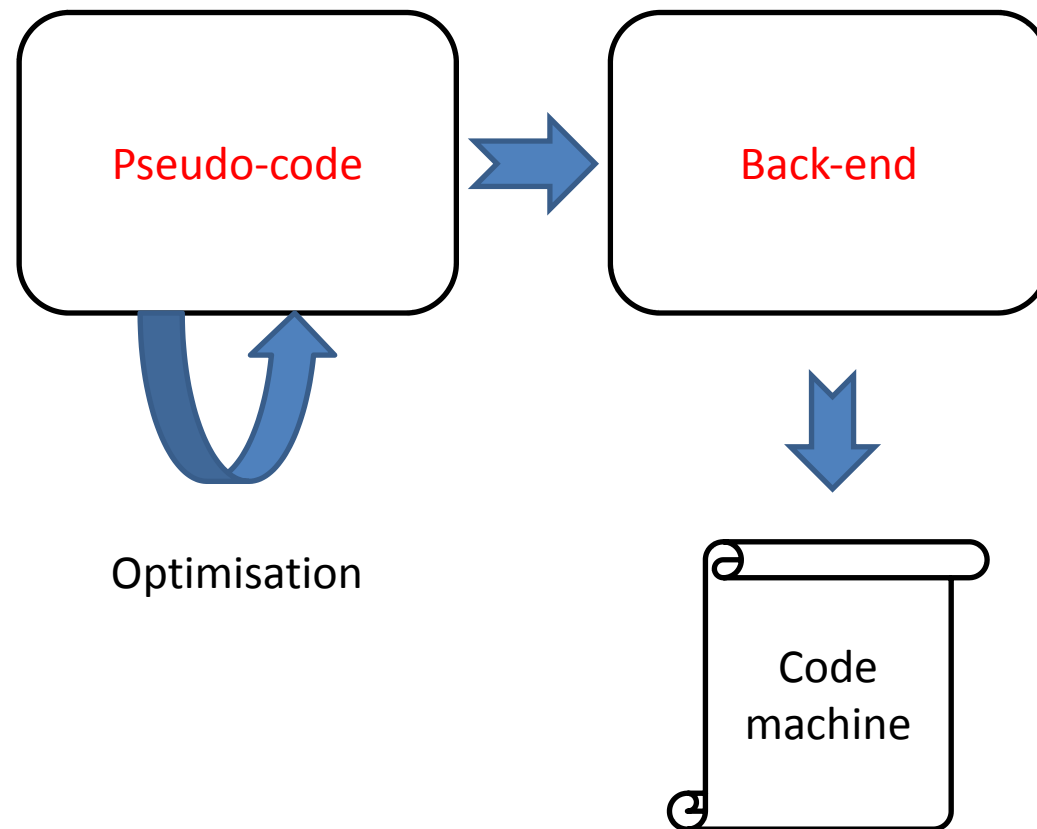
# Etapes de la compilation



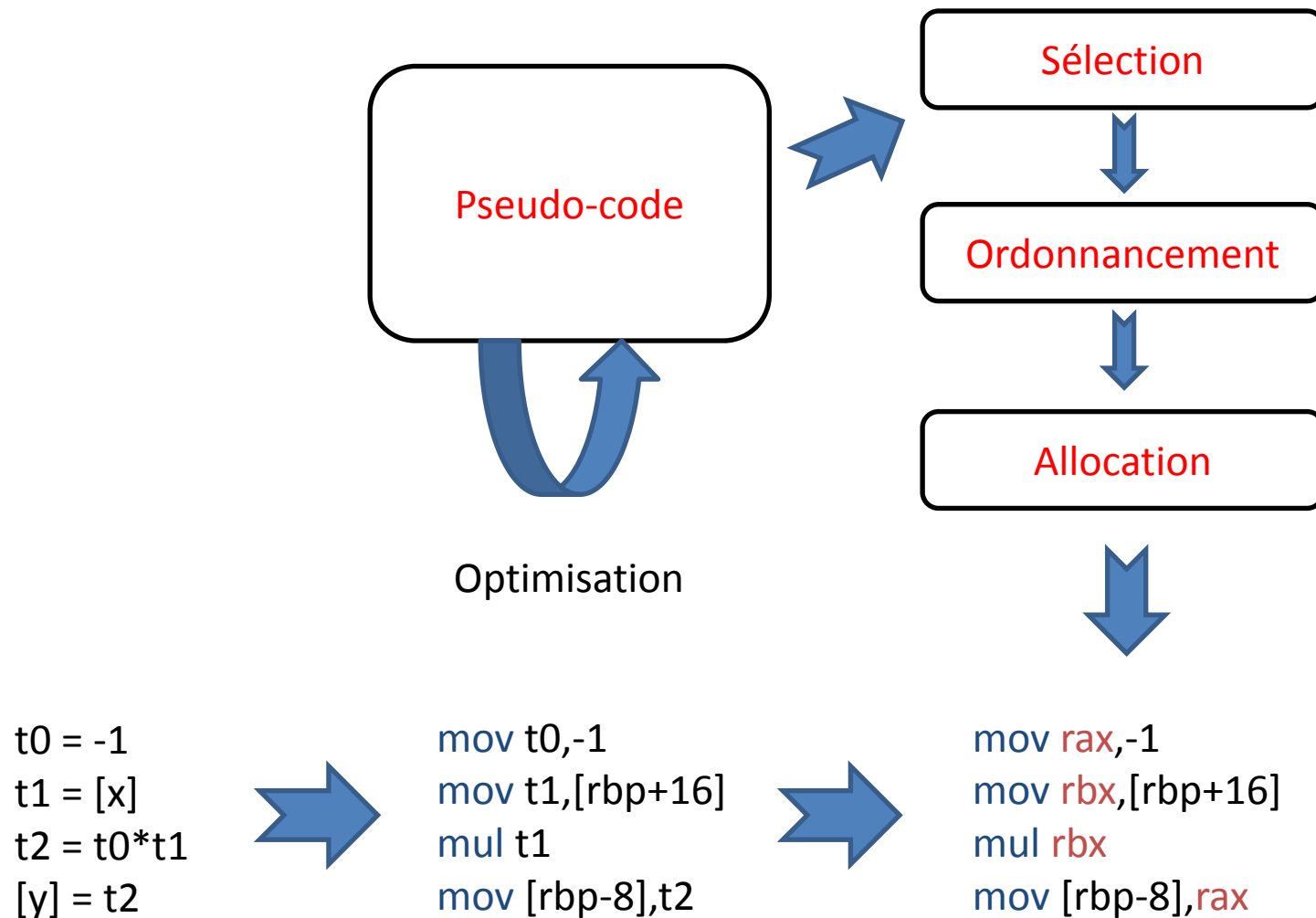
# Etapes de la compilation



# Etapes de la compilation

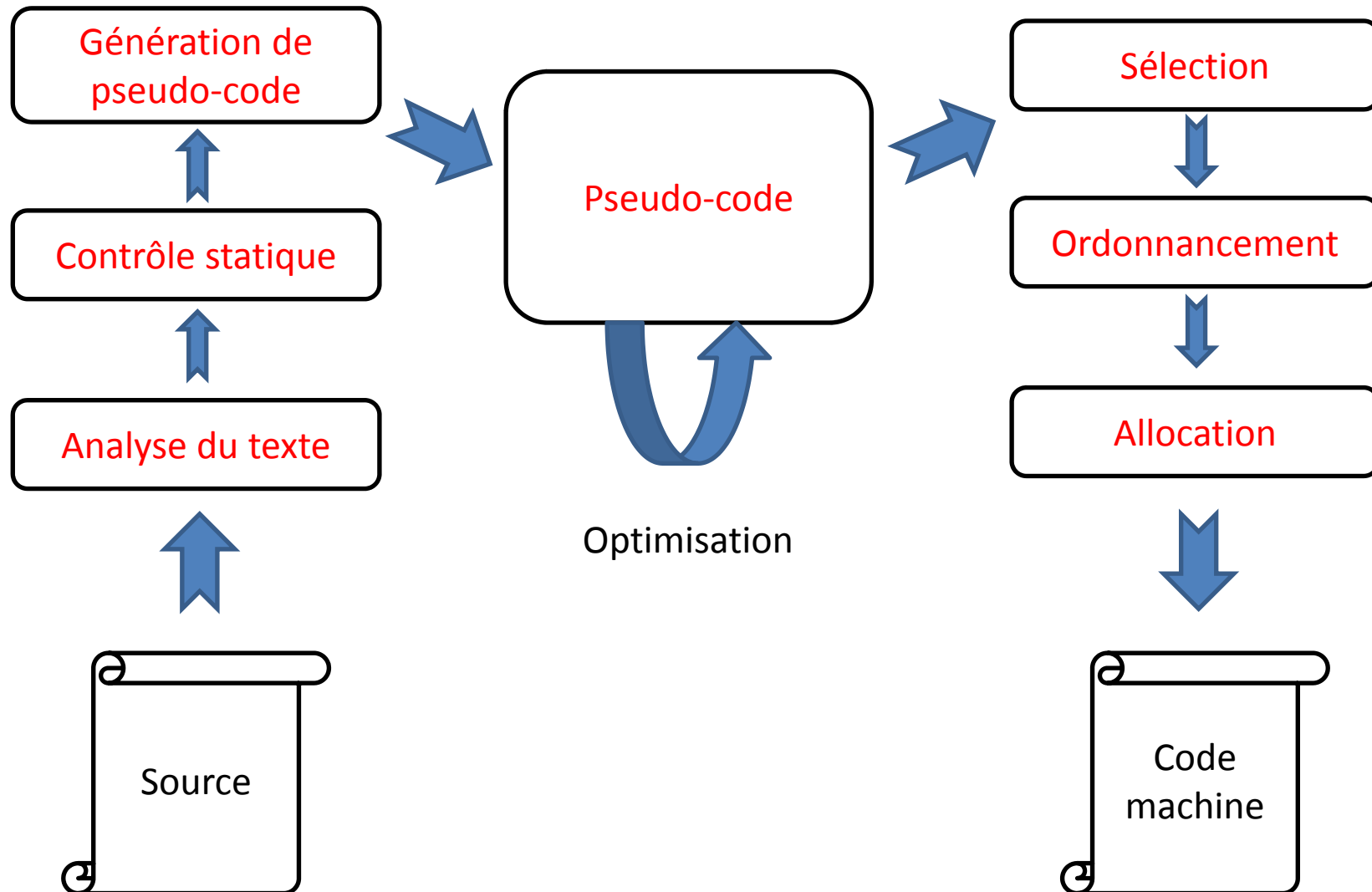


# Etapes de la compilation

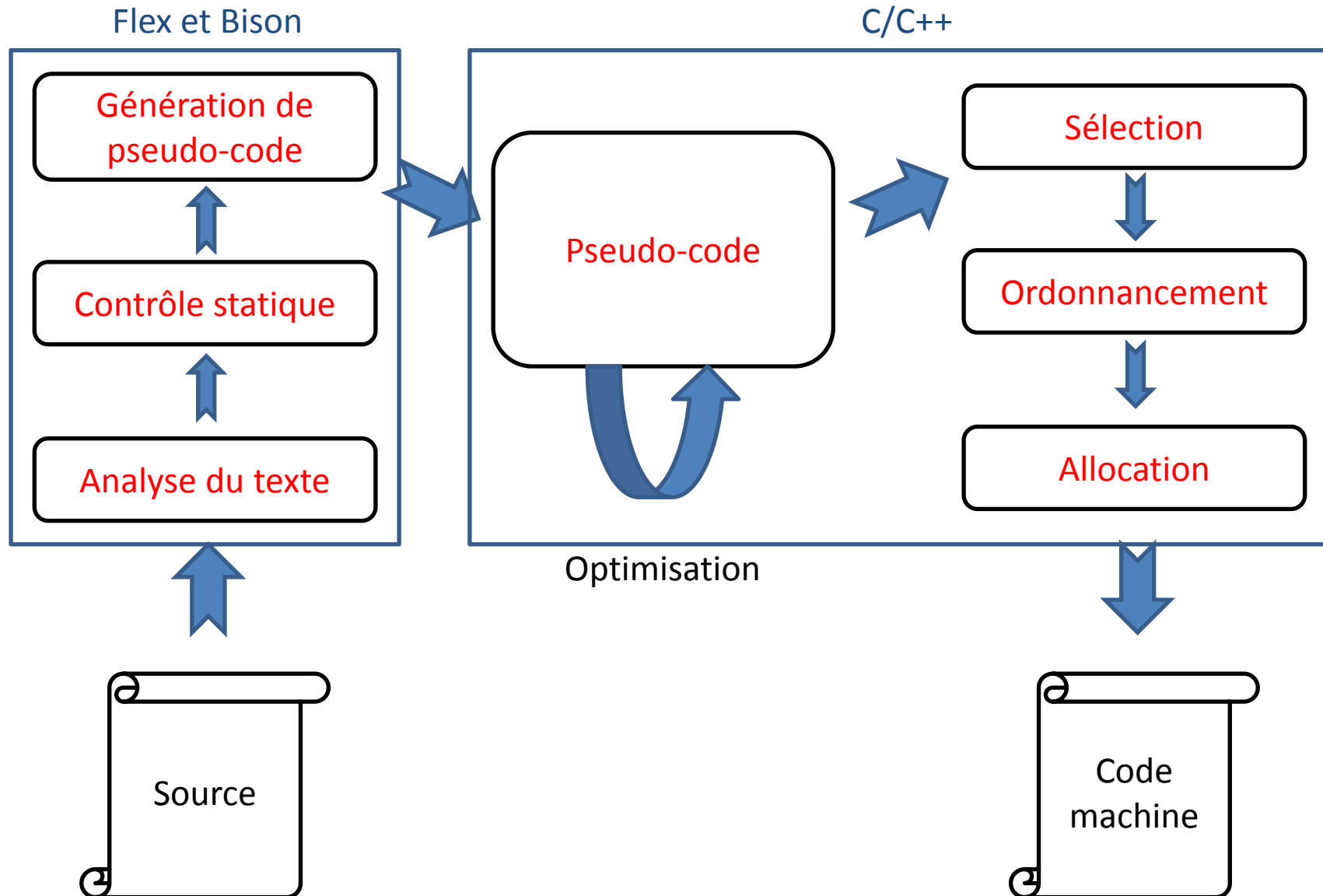




# Etapes de la compilation



# Outils

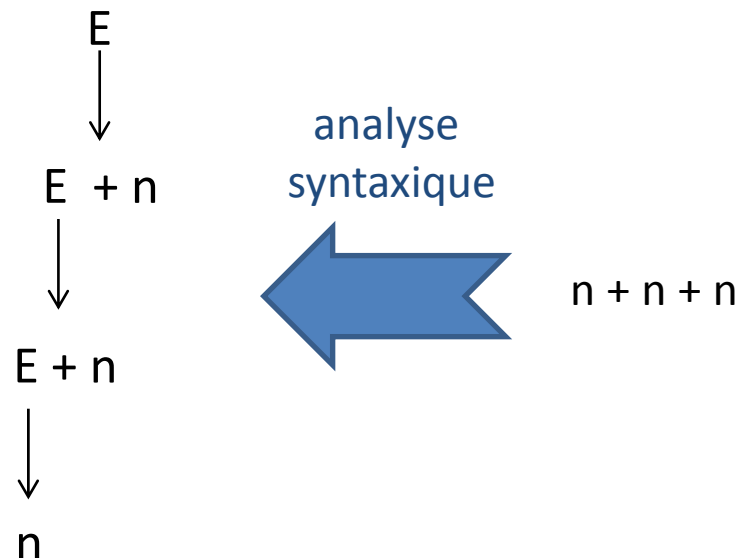


# Analyse du texte (rappels)

- Le langage d'entrée est spécifié par une **grammaire hors-contexte**
- Analyser le texte, c'est **reconstruire** l'arbre de dérivation du texte

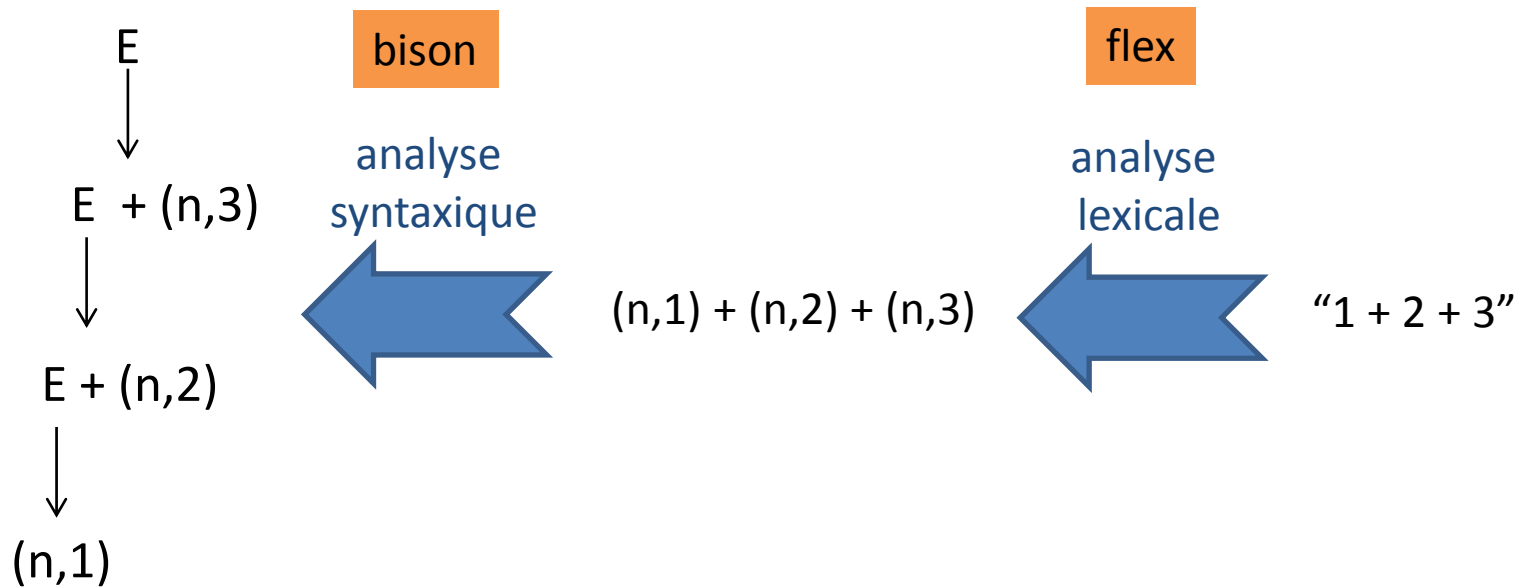
$E \rightarrow E + n$

$E \rightarrow n$

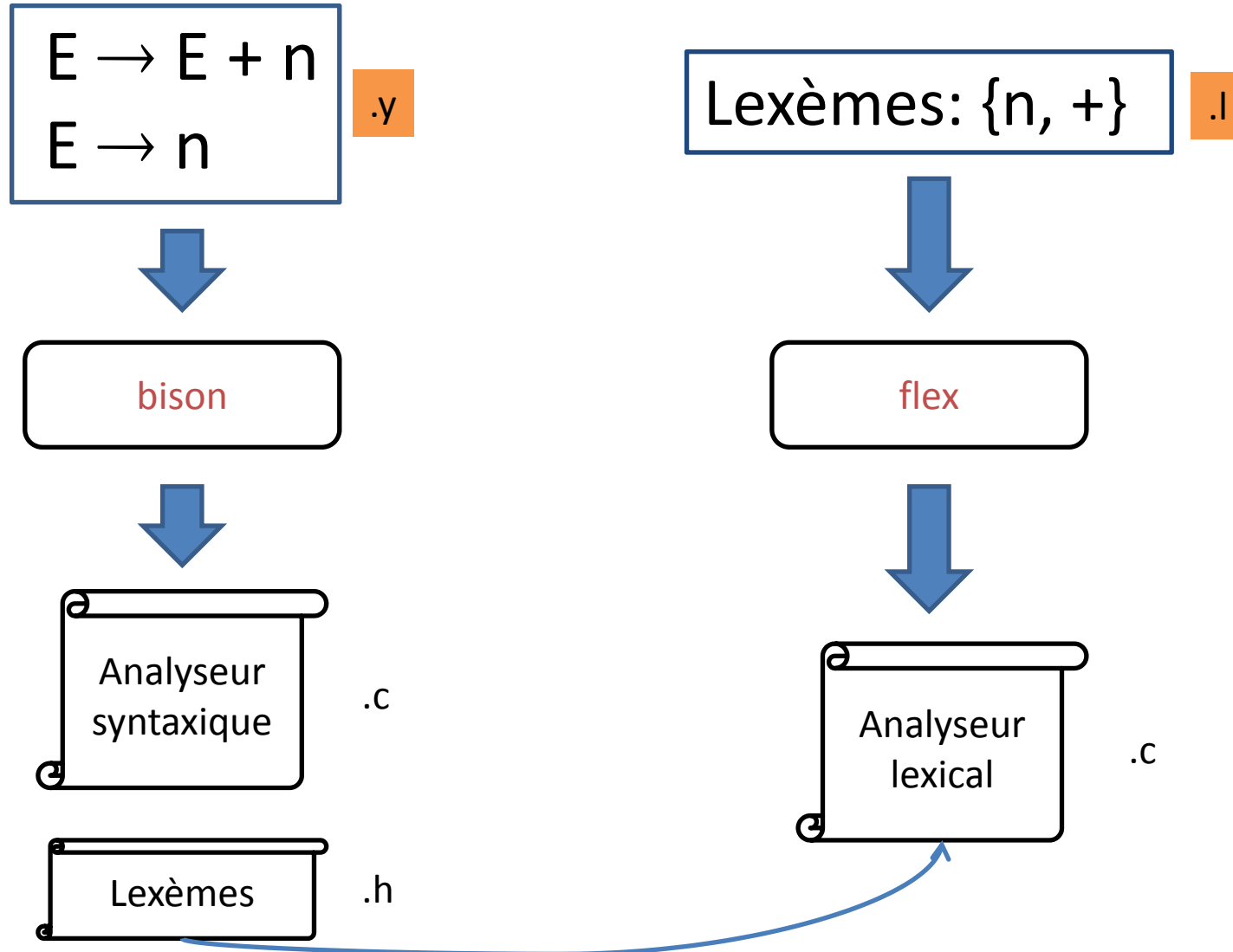


# Analyse lexicale (rappels)

- En amont, on **divise** le texte en lexèmes
- Les lexèmes peuvent être **attribués**



# Flex & Bison (rappels)



# Flex & Bison (rappels)

$E \rightarrow E + n$   
 $E \rightarrow n$

.y

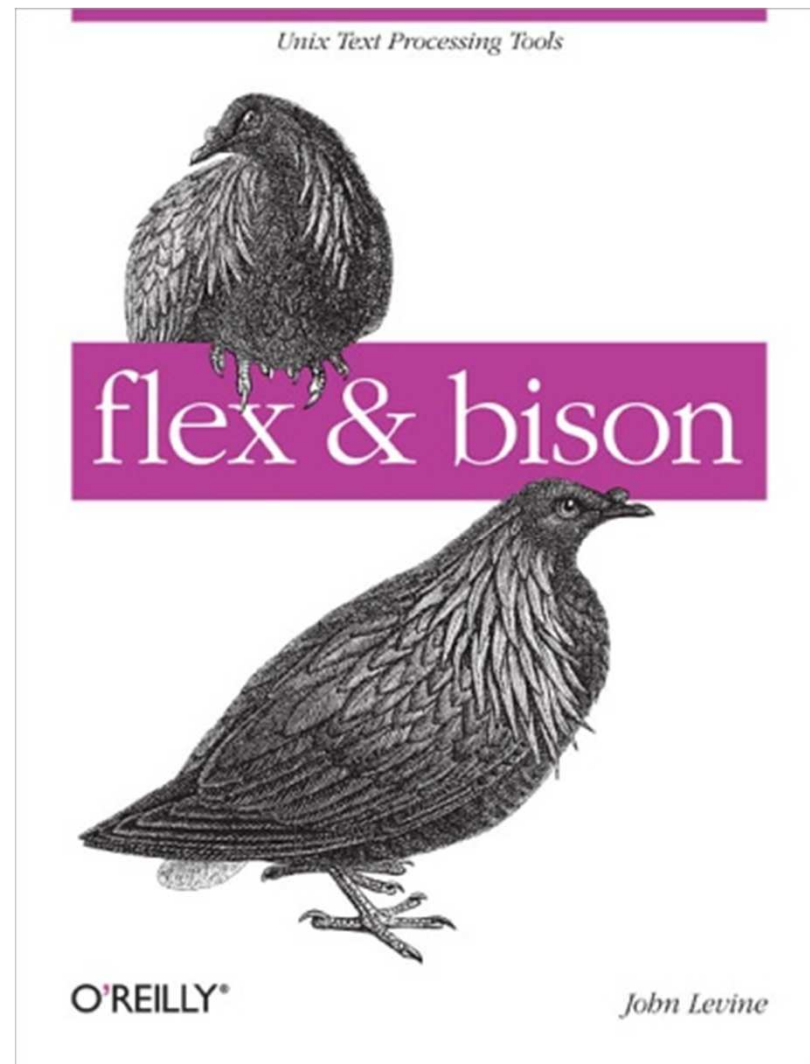
```
%{  
    //Declarations  
}%  
  
%union { int integer; }  
  
%token TK_VAL TK_PLUS  
  
%type<integer> TK_VAL  
  
/* axiom */  
%start e  
  
%%  
  
e: e TK_PLUS TK_VAL  
  | TK_VAL  
  ;
```

Lexèmes: {n, +}

.l

```
%{  
    //Declarations  
}%  
  
%%  
"+"      return TK_PLUS;  
  
[0-9]+   {  
            yylval.integer = atoi(yytext);  
            return TK_VAL;  
        }
```

# Flex & Bison: référence



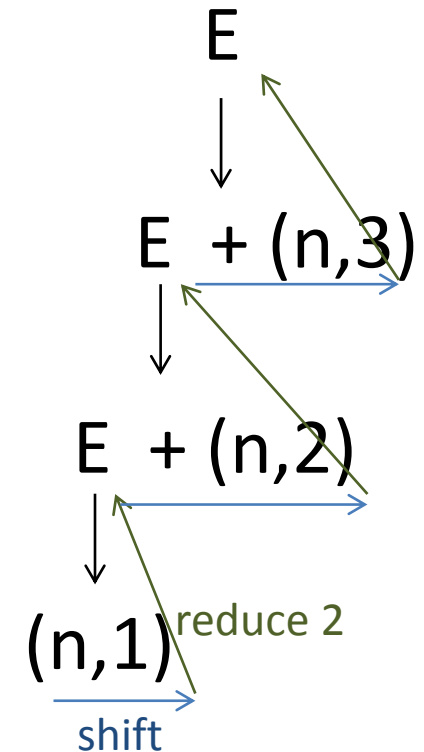
# Analyse ascendante (rappels)

- On lit le flot de lexèmes jusqu'à reconnaître une règle (**shift**)
- On substitue par le non-terminal (**reduce**)

$$E \rightarrow E + n \quad (1)$$

$$E \rightarrow n \quad (2)$$

$(n,1) + (n,2) + (n,3)$   
 $\xrightarrow{\text{shift}}$





# Grammaires attribuées (rappels)

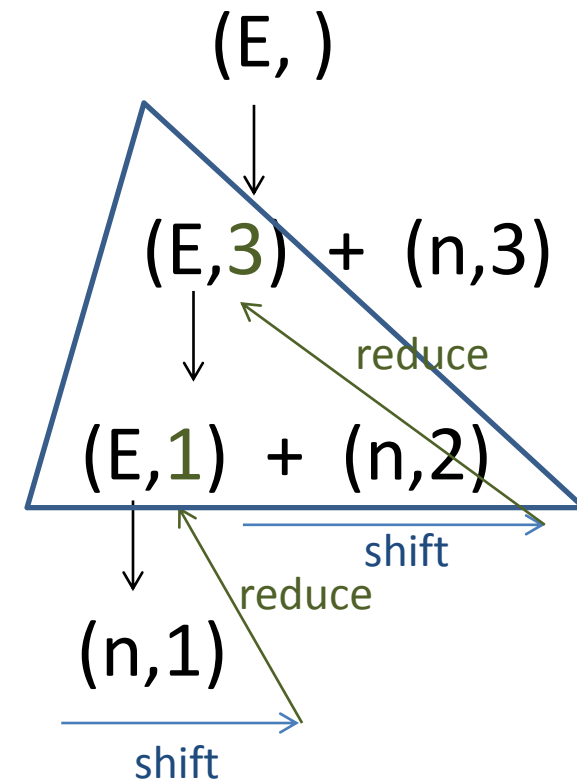
$E \rightarrow E + n \quad \{ \$\$ = \$1 + \$3; \}$

$E \rightarrow n \quad \{ \$\$ = \$1; \}$

%union { int integer; }

%type<integer> n

%type<integer> E

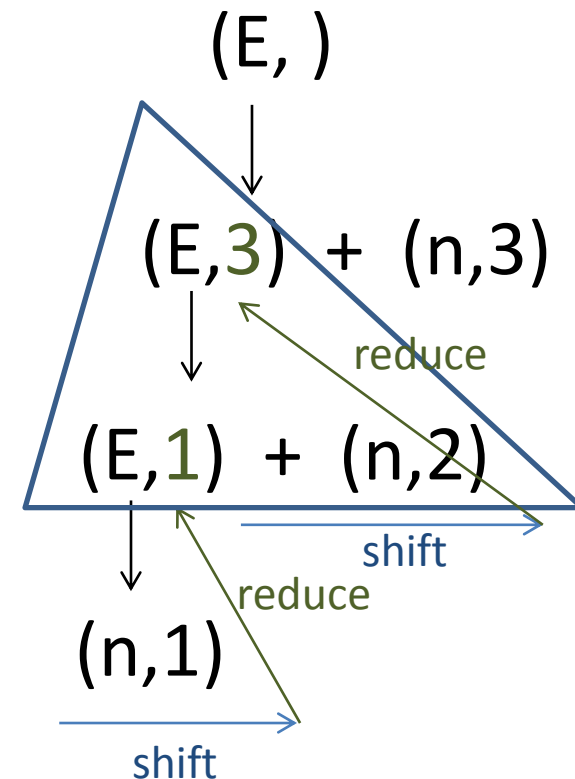


# Grammaires attribuées (rappels)

$E \rightarrow E + n \quad \{ \$\$ = \$1 + \$3; \}$

“peut être” vu comme:

```
rec_E() {  
  $1 = rec_E();  
  $2 = rec("+");  
  $3 = rec_n();  
  { $ = $1 + $3; }  
  return $;  
}
```

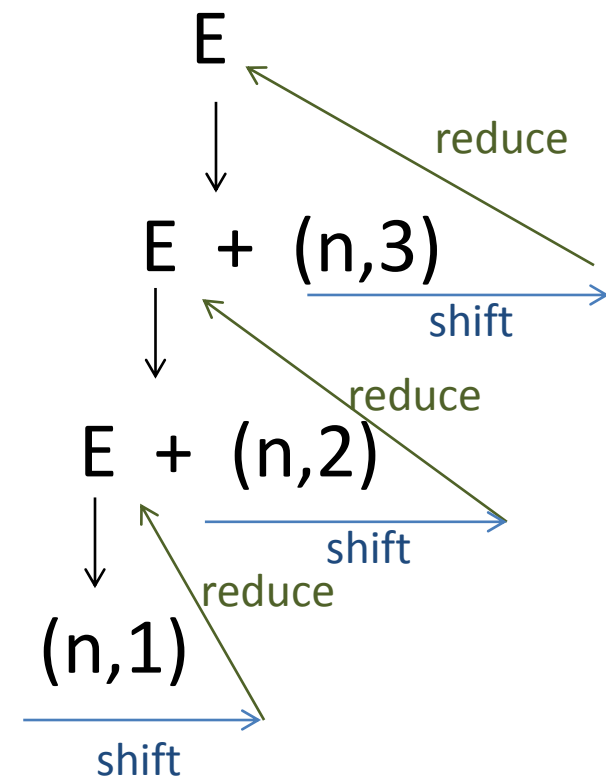


# Quizz

Que fait la grammaire attribuée:

$E \rightarrow E + n \quad \{ \text{cout} \ll \$3 \ll "+"; \}$

$E \rightarrow n \quad \{ \text{cout} \ll \$1; \}$



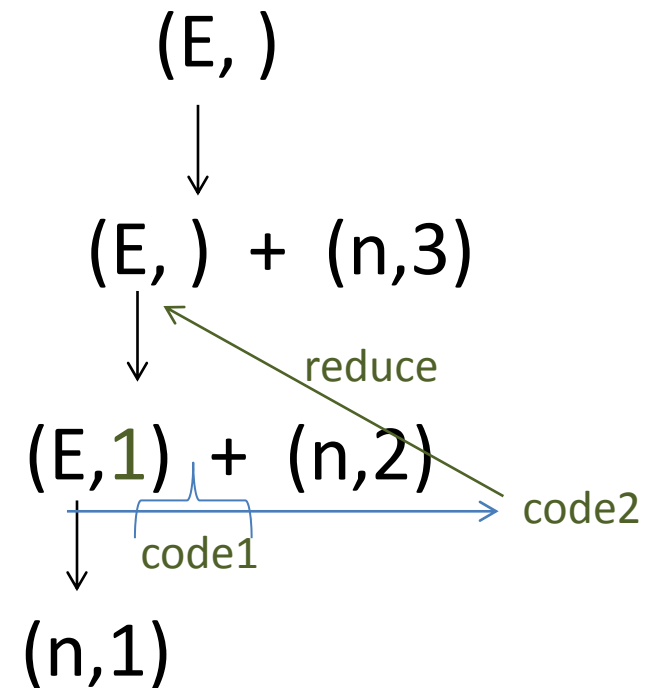
# Schémas de traduction

- On veut exécuter des **actions** pendant la reconnaissance de la règle:

$E \rightarrow E \{ \text{code1} \} + n \{ \text{code2} \}$

- “Peut être vu” comme:

```
rec_E() {  
  $1 = rec_E();  
  { code1 }  
  $3 = rec("+");  
  $4 = rec_n();  
  { code2 }  
}
```

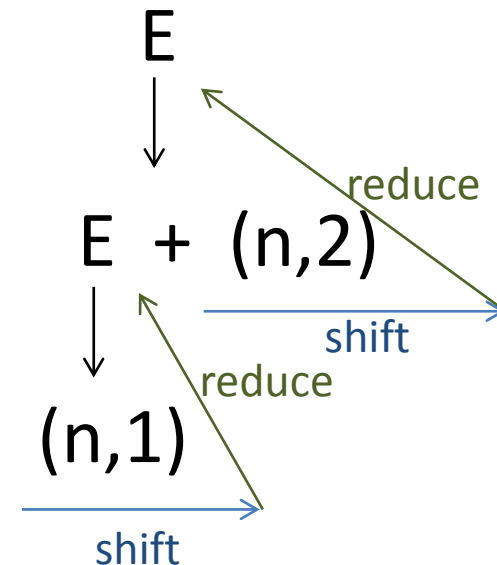


# Quizz

Que fait la grammaire attribuée:

$E \rightarrow \{p(\text{"+"});\} E \{p(\text{","});\} + n \{p(\$5); p(\text{"})");\}$

$E \rightarrow n \{p(\$1);\}$



# Traduction dirigée par la syntaxe

- Traduction au fur et à mesure de l'analyse
- Règle de traduction pour chaque production de la grammaire:

$E \rightarrow E + n$  { cout << \$3 << "+"; }

$E \rightarrow n$  { cout << \$1; }

# Opérateurs de traduction

$\llbracket E \rrbracket$  = traduction de E

- $\llbracket E + n \rrbracket = \llbracket E \rrbracket \textcolor{red}{n} +$
- $\llbracket n \rrbracket = \textcolor{red}{n}$

# Quizz

Dérouler  $\llbracket 1 + 2 + 3 \rrbracket$



# Exemple

Machine à pile (**mpush**, **add**)

$\llbracket E \rrbracket$  évalue  $E$  et place le résultat sur la pile

- $\llbracket n \rrbracket =$   
**mpush**  $n$
- $\llbracket E + n \rrbracket =$   
 $\llbracket E \rrbracket$   
**mpush**  $n$   
**add**

Implémentation

$E \rightarrow E + n$	$\{ \text{mpush}(\$3); \text{add}(); \}$
$E \rightarrow n$	$\{ \text{mpush}(\$1); \}$

# Quizz

Traduire “1+2+3”